ProtoDUNE SP Analysis Utilities

Leigh Whitehead

ProtoDUNE Analysis Workshop

27/01/19

UNIVERSITY OF CAMBRIDGE

Introduction

- There are a number of utility classes that are available to help simplify analysis code

- A lot of the tools are to help users extract information from the Pandora output in the recommended fashion

- Tools also exist to help with extraction of information from data files (beam trigger etc)

- The tools can all be found here:
  - dunetpc/dune/Protodune/Analysis/
  - dunetpc redmine link

Need to link the library:
ProtoDUNEAnaUtils

List of Utilities

- There are currently six classes for different objects:

ProtoDUNE data	ProtoDUNEDataUtils
MC Truth	ProtoDUNETruthUtils
Reconstructed Tracks	ProtoDUNETrackUtils
Reconstructed Showers	ProtoDUNEShowerUtils
Reconstructed Slices	ProtoDUNESliceUtils
Reconstructed PFParticles	ProtoDUNEPFParticleUtils

- The header files are currently the best documentation for which functions exist

- I will cover some highlights of the ones I think are most useful
    - A focus on the PFParticles as these should be the objects used in analyses

NB: many of these functions in the coming slides make use of the pandora particle metadata

PFParticles

- One of the most important things for ProtoDUNE analyses is knowing which particle comes from the beam

By default, this is "pandora"

  - Is a given particle the beam particle?

```
/// Use the pandora metadata to tell us if this is a beam particle or not
bool IsBeamParticle(const recob::PFParticle &particle, art::Event const &evt, const std::string particleLabel) const;
```

  - Alternatively, we can request all primary particles in the beam slice

```
/// Return the pointers for the PFParticles in the beam slice. Returns an empty vector is no beam slice was found
const std::vector<const recob::PFParticle*> GetPFParticlesFromBeamSlice(art::Event const &evt, const std::string particleLabel) const;
```

  - If we just want to know the slice containing the beam particle

```
/// Try to get the slice tagged as beam. Returns 9999 if no beam slice was found
unsigned short GetBeamSlice(art::Event const &evt, const std::string particleLabel) const;
```

  - We can also get the BDT score that decides between beam / cosmic

```
/// Access the BDT output used to decide if a slice is beam-like or cosmic-like
float GetBeamCosmicScore(const recob::PFParticle &particle, art::Event const &evt, const std::string particleLabel) const;
```

PFParticles – Properties

- We may want to know about our particle
  - Is this particle track-like or shower-like?

```
/// Is the particle track-like?
bool IsPFParticleTracklike(const recob::PFParticle &particle) const;

/// Is the particle track-like?
bool IsPFParticleShowerlike(const recob::PFParticle &particle) const;
```

  - Is this particle one of the clear cosmics?

```
/// Pandora tags and removes clear cosmics before slicing, so check if this particle is a clear cosmic
bool IsClearCosmic(const recob::PFParticle &particle, art::Event const &evt, const std::string particleLabel) const;
```

  - Does the particle have a reconstructed T0?

```
/// Get the T0(s) from a given PFParticle
std::vector<anab::T0> GetPFParticleT0(const recob::PFParticle &particle, art::Event const &evt, std::string particleLabel) const;
```

This vector will be empty if there is no T0, otherwise it will have element containing the measured T0

PFParticles – Associated Products

- Get associated objects (encapsulates the art associations)
  - Get the track or shower from the PFParticle

By default, this is “pandoraTrack”

```
/// Get the track associated to this particle. Returns a null pointer if not found.
const recob::Track* GetPFParticleTrack(const recob::PFParticle &particle, art::Event const &evt, const std::string particleLabel, const std::string trackLabel) const;

/// Get the shower associated to this particle. Returns a null pointer if not found.
const recob::Shower* GetPFParticleShower(const recob::PFParticle &particle, art::Event const &evt, const std::string particleLabel, const std::string showerLabel) const;
```

By default, this is “pandoraShower”

  - Get the clusters

```
/// Get the clusters associated to the PFParticle
const std::vector<const recob::Cluster*> GetPFParticleClusters(const recob::PFParticle &particle, art::Event const &evt, const std::string particleLabel) const;
```

  - Get the space points

```
// Get the SpacePoints associated to the PFParticle
const std::vector<const recob::SpacePoint*> GetPFParticleSpacePoints(const recob::PFParticle &particle, art::Event const &evt, const std::string particleLabel) const;
```

  - Get the hits

```
/// Get the hits associated to the PFParticle
const std::vector<const recob::Hit*> GetPFParticleHits(const recob::PFParticle &particle, art::Event const &evt, const std::string particleLabel) const;
```

PFParticles – Vertices

- There has been some recent discussion about particle vertices
  - These functions might be updated soon, but the current status is a follows:
  - Particle vertex means the start point of the object

```
/// Function to find the interaction vertex of a primary PFParticle
const TVector3 GetPFParticleVertex(const recob::PFParticle &particle, art::Event const &evt, const std::string particleLabel, const std::string trackLabel) const;
```

    - Showers: this comes from from the recob::Vertex associated to the PFParticle
    - Beam tracks: most upstream end of the recob::Track
    - Cosmic tracks: end of the recob::Track with the larger y-coordinate
  - Secondary vertex means the interaction point

```
/// Function to find the secondary interaction vertex of a primary PFParticle
const TVector3 GetPFParticleSecondaryVertex(const recob::PFParticle &particle, art::Event const &evt, const std::string particleLabel, const std::string trackLabel) const;
```

    - Showers: returns a dummy vector as showers have no secondary vertex
    - Beam tracks: most downstream end of the recob::Track
    - Cosmic tracks: end of the recob::Track with the smaller y-coordinate

We are currently trying to optimise this information within Pandora and these tools, so things are subject to change, but hopefully in an invisible way to the user

Data Utilities

- The data utilities mostly help with trigger information and beam PID
  - Thanks to Justin for putting most of this together
  - Simple beam trigger check (using the CTB)

```
/**
 * Returns true if the ProtoDUNE trigger says this is a beam trigger
 */
bool IsBeamTrigger(art::Event const & evt) const;
```

  - Beamline beam trigger check

```
/**
 * Returns true if the beamline instrumentation has a good trigger
 * that matches the ProtoDUNE trigger.
 */
bool IsGoodBeamlineTrigger(art::Event const & evt) const;
```

  - Check if we had all fembs working in a given APA

```
/// Get number of active fembs in an APA
int GetNActiveFembsForAPA(art::Event const & evt, int apa) const;
```

Data Utilities

- The data utilities mostly help with trigger information and beam PID
  - Thanks to Justin for putting most of this together
- There are a number of very important functions for accessing the beamline PID and TOF information
  - Please use these functions!
- Too many to list here, but they are documented in the header file
  - dunetpc/dune/Protodune/Analysis/ProtoDUNEDataUtils.h

A Usage Example

- There is a skeleton module you can use to get started
  - dunetpc/dune/Protodune/Analysis/BeamExample/BeamExample_module.cc

- It runs on data and MC and selects the reconstructed beam particle and extracts some information

```
bool beamTriggerEvent = false;
// If this event is MC then we can check what the true beam particle is
if(!evt.isRealData()){
  // Get the truth utility to help us out
  protoana::ProtoDUNETruthUtils truthUtil;
  // Firstly we need to get the list of MCTruth objects from the generator. The standard protoDUNE
  // simulation has fGeneratorTag = "generator"
  auto mcTruths = evt.getValidHandle<std::vector<simb::MCTruth>>(fGeneratorTag);
  // mcTruths is basically a pointer to an std::vector of simb::MCTruth objects. There should only be one
  // of these, so we pass the first element into the function to get the good particle
  const simb::MCParticle* geantGoodParticle = truthUtil.GetGeantGoodParticle((*mcTruths)[0],evt);
  if(geantGoodParticle != 0x0){
    std::cout << "Found GEANT particle corresponding to the good particle with pdg = " << geantGoodParticle->PdgCode() << std::endl;
  }
}
else{
  // For data we can see if this event comes from a beam trigger
  beamTriggerEvent = dataUtil.IsBeamTrigger(evt);
  if(beamTriggerEvent){
    std::cout << "This data event has a beam trigger" << std::endl;
  }
}
```

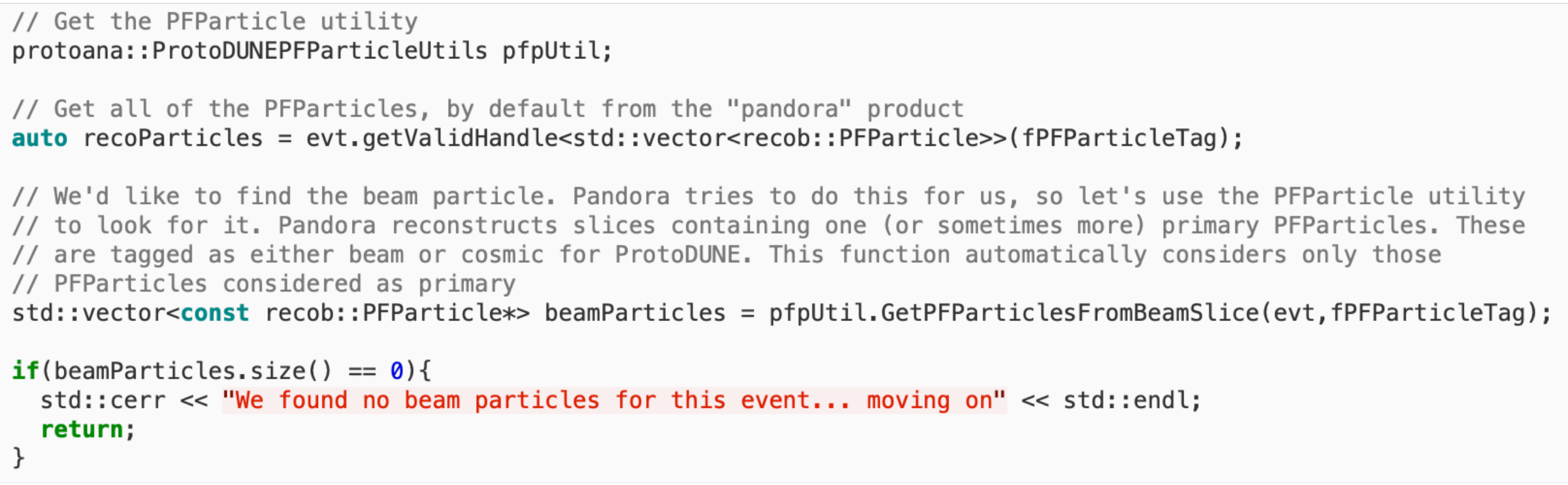
For MC, it finds the triggered true particle using the truth utility

For Data, it looks for a beam trigger using the data utility

A Usage Example

- There is a skeleton module you can use to get started
  - dunetpc/dune/Protodune/Analysis/BeamExample/BeamExample_module.cc

- It then looks for the beam PFParticles

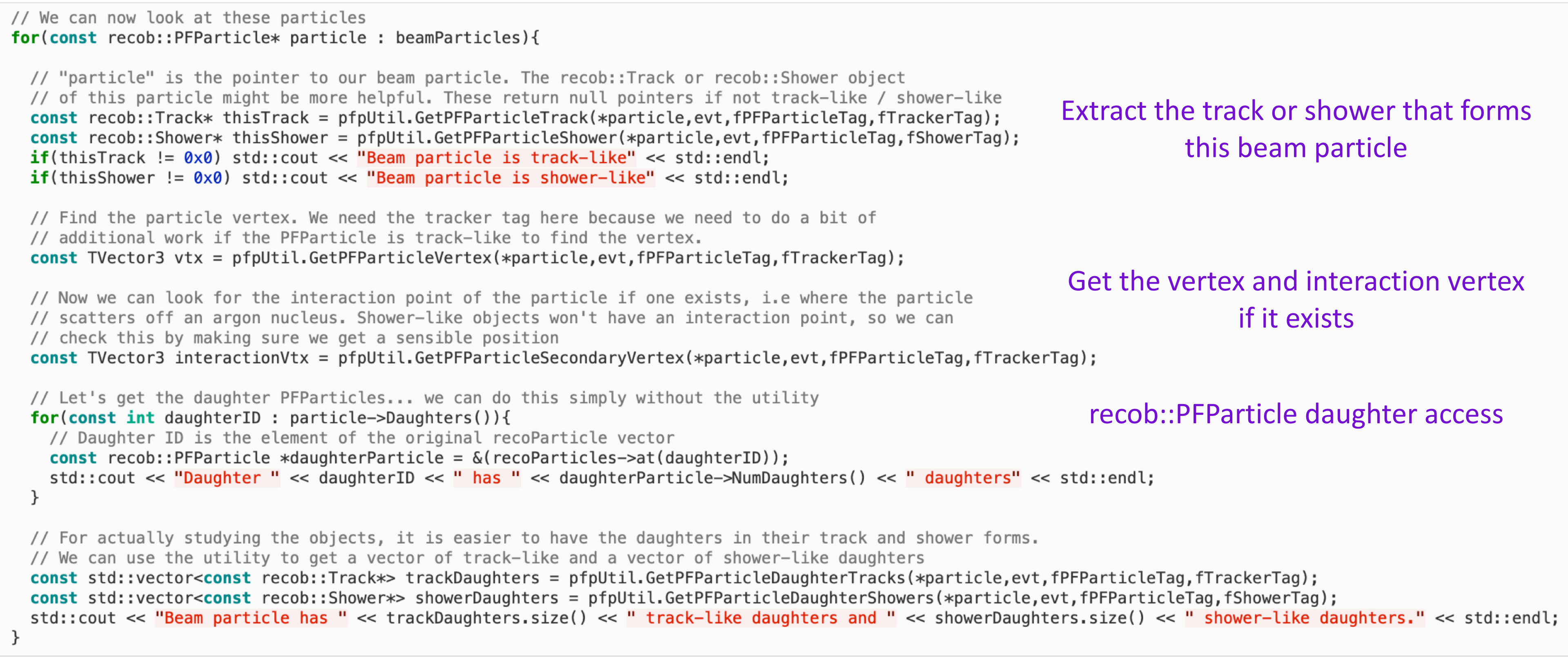


```
// Get the PFParticle utility
protoana::ProtoDUNEPFParticleUtils pfpUtil;

// Get all of the PFParticles, by default from the "pandora" product
auto recoParticles = evt.getValidHandle<std::vector<recob::PFParticle>>(fPFParticleTag);

// We'd like to find the beam particle. Pandora tries to do this for us, so let's use the PFParticle utility
// to look for it. Pandora reconstructs slices containing one (or sometimes more) primary PFParticles. These
// are tagged as either beam or cosmic for ProtoDUNE. This function automatically considers only those
// PFParticles considered as primary
std::vector<const recob::PFParticle*> beamParticles = pfpUtil.GetPFParticlesFromBeamSlice(evt,fPFParticleTag);

if(beamParticles.size() == 0){
  std::cerr << "We found no beam particles for this event... moving on" << std::endl;
  return;
}
```

Here we use the PFParticle utility to get all of the primary particles in the beam slice (typically just one)

A Usage Example

- There is a skeleton module you can use to get started
  - dunetpc/dune/Protodune/Analysis/BeamExample/BeamExample_module.cc



```
// We can now look at these particles
for(const recob::PFParticle* particle : beamParticles){

  // "particle" is the pointer to our beam particle. The recob::Track or recob::Shower object
  // of this particle might be more helpful. These return null pointers if not track-like / shower-like
  const recob::Track* thisTrack = pfpUtil.GetPFParticleTrack(*particle,evt,fPFParticleTag,fTrackerTag);
  const recob::Shower* thisShower = pfpUtil.GetPFParticleShower(*particle,evt,fPFParticleTag,fShowerTag);
  if(thisTrack != 0x0) std::cout << "Beam particle is track-like" << std::endl;
  if(thisShower != 0x0) std::cout << "Beam particle is shower-like" << std::endl;

  // Find the particle vertex. We need the tracker tag here because we need to do a bit of
  // additional work if the PFParticle is track-like to find the vertex.
  const TVector3 vtx = pfpUtil.GetPFParticleVertex(*particle,evt,fPFParticleTag,fTrackerTag);

  // Now we can look for the interaction point of the particle if one exists, i.e where the particle
  // scatters off an argon nucleus. Shower-like objects won't have an interaction point, so we can
  // check this by making sure we get a sensible position
  const TVector3 interactionVtx = pfpUtil.GetPFParticleSecondaryVertex(*particle,evt,fPFParticleTag,fTrackerTag);

  // Let's get the daughter PFParticles... we can do this simply without the utility
  for(const int daughterID : particle->Daughters()){
    // Daughter ID is the element of the original recoParticle vector
    const recob::PFParticle *daughterParticle = &(recoParticles->at(daughterID));
    std::cout << "Daughter " << daughterID << " has " << daughterParticle->NumDaughters() << " daughters" << std::endl;
  }

  // For actually studying the objects, it is easier to have the daughters in their track and shower forms.
  // We can use the utility to get a vector of track-like and a vector of shower-like daughters
  const std::vector<const recob::Track*> trackDaughters = pfpUtil.GetPFParticleDaughterTracks(*particle,evt,fPFParticleTag,fTrackerTag);
  const std::vector<const recob::Shower*> showerDaughters = pfpUtil.GetPFParticleDaughterShowers(*particle,evt,fPFParticleTag,fShowerTag);
  std::cout << "Beam particle has " << trackDaughters.size() << " track-like daughters and " << showerDaughters.size() << " shower-like daughters." << std::endl;
}
```

Extract the track or shower that forms this beam particle

Get the vertex and interaction vertex if it exists

recob::PFParticle daughter access

Get the track and shower objects corresponding to the daughter particles

Summary

- I hope that these tools can help to have a unified approach for accessing the important information for ProtoDUNE analyses

- I haven’t been able to cover everything that’s included in the tools but I’d invite anyone starting an analysis to take a look
  - It’ll save time and effort if the information you need is already available somewhere

- These tools are by no means complete!
  - You should all feel free to suggest new features or modifications to the current ones to be more useful for your given use-case
  - One of these I know is to return art::Ptr objects… I will look into having functions to return these as well as const recob::Object pointers